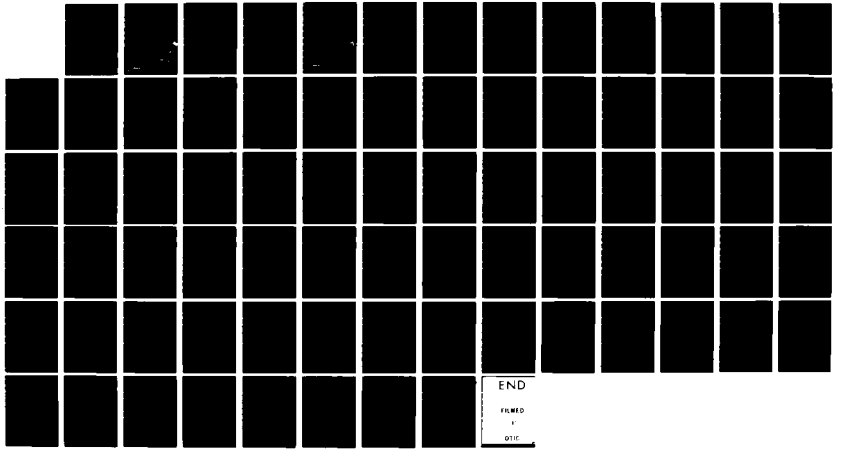
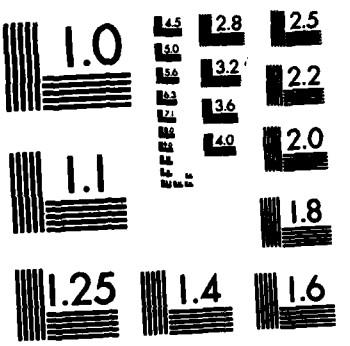
AD-A124 525 IMPLEMENTING MESSAGE SYSTEMS IN MULTILEVEL SECURE ENVIRONMENTS: PROBLEMS AND APPROACHES(U) RAND CORP SANTA MONICA CA G R MARTINS ET AL. JUL 82 1/1

UNCLASSIFIED RAND/N-1852-DNA DNA001-79-C-0201 F/G 17/2 NL





(12)

AD A121525

A RAND NOTE

IMPLEMENTING MESSAGE SYSTEMS IN MULTILEVEL SECURE ENVIRONMENTS: PROBLEMS AND APPROACHES

Gary R. Martins
R. Stockton Gaines

July 1982

N-1852-DNA

Prepared for

The Defense Nuclear Agency

Rand
SANTA MONICA, CA 90406

The research described in this report was sponsored by the Defense Nuclear Agency under Contract No. DNA001-79-C-0201.

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: N-1852-DNA	2. GOVT ACCESSION NO.: AD-A114 325
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): IMPLEMENTING MESSAGE SYSTEMS IN MULTILEVEL SECURE ENVIRONMENTS: PROBLEMS AND APPROACHES	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER: N-1852-DNA
7. AUTHOR(s): Gary R. Martins	8. CONTRACT OR GRANT NUMBER(s): DNA001-79-C-0201
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, California 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Nuclear Agency, Washington, D.C. 20305	12. REPORT DATE: July 1982
	13. NUMBER OF PAGES: 70
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Secure Communication
Computer Systems Programs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A study of the problems of building multilevel secure message systems. The need for such systems in the government and commercial sectors is growing. Designs are strongly affected by (1) the granularity of security protection (at the level of folders, messages, paragrahps, or words) and (2) planned departures from the Bell-LaPadula security model, for user convenience. A Taxonomy of design alternatives is defined, and sixteen specific approaches are described and compared.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A RAND NOTE

IMPLEMENTING MESSAGE SYSTEMS IN MULTILEVEL SECURE ENVIRONMENTS: PROBLEMS AND APPROACHES

Gary R. Martins
R. Stockton Gaines

July 1982

N-1852-DNA

Prepared for

The Defense Nuclear Agency

Rand
SANTA MONICA, CA. 90406

PREFACE

The development of prototype multilevel secure operating systems has been a significant advance in the field of computer security. Automated message handling systems represent a major application for such systems. However, the requirement that users be able to conveniently process streams and collections of messages at different security levels may collide with constraints imposed by an operating system's attempts to separate the processing of classified data strictly by security level.

This Note, prepared under Contract No. DNA001-79-C-0201 for the Defense Nuclear Agency, deals with certain aspects of security-policy/technology interactions in message system designs for multilevel secure environments. The work reported here focuses primarily on kernel-based secure operating systems (KSOS) and two unique message system architectures developed at Rand (MH and SMH).

The Note examines the conflict between requirements and outlines a number of design alternatives that might resolve or ameliorate it. In addition, a number of basic security issues in systems design--such as granularity of security levels and downgrading--are reviewed. Thus, the study should be of use to message systems designers and to designers of related applications software. It should also be of interest to policy decisionmakers concerned with security issues, and to persons who wish to procure message systems for secure environments.

SUMMARY

Highly capable secure computer operating systems will be generally available in the near future, and multilevel secure message services will be needed in many applications of such systems. The design of multilevel secure message systems involves tradeoffs among security requirements, functionality, convenience, technical feasibility, cost, and other factors. This Note focuses principally on the interaction between security policy and technical design in the implementation of multilevel secure message systems. For example, such systems must violate a common current computer implementation of the DoD security policy in order to achieve an acceptable level of functionality. The basic properties of message systems are briefly reviewed, several key technical issues are discussed (including the concepts of granularity and trust), and a variety of possible design approaches are outlined and informally evaluated. These approaches are based upon studies of KSOS (a prototype of which, KSOS-11, has been implemented and is being evaluated) and two Rand-developed message systems, MH and SMH.

CONTENTS

I. INTRODUCTION

As the interactive use of computers increases, the need for security in computer systems becomes more widely recognized. Current developments in operating systems security will soon provide substantially more powerful security capabilities (Walker, 1980). But little is known about how to best employ such capabilities in applications software. This Note focuses on one important application for secure computer systems, multilevel secure message systems.

Computer-based message systems are increasingly becoming essential components of military command center support; they have been a major application of computers within the intelligence community for several years. In these contexts, messages of many different classifications must be dealt with simultaneously in a way that both ensures the protection appropriate to the different security levels and makes it convenient for authorized personnel to deal with those messages. This Note reviews some of the technical and policy issues associated with designing a multilevel message system to operate within a secure computing system. Many of the security issues that arise in building message systems also arise in other applications of secure systems (database systems, etc.). We therefore anticipate that this study will contribute to a variety of secure computer applications.

There has been very little analytical study of security issues, yet security policy considerations will influence the directions of technology development and application. To date, most technological developments in internal computer system security have been driven by

fairly simple ideas about what security policies must be enforced by secure computer systems. The result, in several cases, has been unnecessarily elaborate and unwieldy designs, since technologists designing the systems have not challenged any of the security requirements or even examined in depth the nature of the policy whose details they were commissioned to implement. This Note attempts to illustrate how security policy considerations may drive the process of systems design.

The goal of any secure computing system is to forbid access to classified data by unauthorized users or unauthorized user programs. Systems are designed to implement a security policy or model which, if rigorously enforced, can guarantee no improper leakage of classified data. The basic security model that underlies today's secure systems (Bell and LaPadula, 1974) may be paraphrased as follows:

1. Every user, file, and program possesses a single level of security clearance.
2. A user may not invoke a process with a higher level of classification than his own security clearance (e.g., a user cleared to the Confidential level may not invoke a program that is cleared to the Secret level).
3. A process at a given clearance level may not <u>read</u> data classified at a higher level (e.g., a Secret-level process may not read Top Secret data).
4. A process at a given clearance level may not <u>write</u> data at a lower level (e.g., a Secret-level process may not write a Confidential file).

This model is simple and unambiguous, and its rigorous implementation will indeed achieve the goals of data security. At the same time, however, it is crude and inflexible; as will be seen, its absolute separation of levels can be an obstacle to the functionality required in various applications, including multilevel message systems. Expansions of the model to include data integrity have been proposed by Biba (1975) and Dion (1981).

The implementation technique of present multilevel secure operating systems designs such as KSOS (McCauley et al., 1978, 1979) is to formally verify a "kernel" of the operating system, proving that it faithfully enforces an underlying security model and protects the security of data. The resulting software is then termed trusted; i.e., it has undergone some certification or verification process. The user's application domain consists of untrusted applications software built upon the kernel. However, as will be illustrated later, applications that must process data of varying security levels (e.g., a multilevel message handling system) may need to violate the basic security model to some degree if they are to be useful. This conflict can be resolved by developing additional "trusted" software which may then violate the basic security model in specific, limited ways; but this additional software must also be certified or verified, often at considerable cost. A key task for the applications software designer, then, is to determine how to least violate the basic security model, limit the amount of additional trusted code, and yet achieve the functionality demanded by the application.

Current message system technology generally divides message system functions into (1) operating system functions (the "mailer") for distributing messages within the system and to and from networks and (2) interactive programs that allow the user to manipulate (e.g., read, save, and compose) a collection of messages. In this Note we focus mainly on the second set of functions. Implementation of the first set in a multilevel secure operating system is discussed in McCauley et al. (1979). Security aspects of the second set of functions are exained in Landwehr (1980) and Miller and Resnick (1981).

It should be emphasized that little experience exists in dealing with messages at different levels simultaneously on the same computer. The National Military Indications Center and the National Military Command Center (NMCC) have been using a system which permits messages to be read and retrieved on-line using a CRT terminal (Defense Intelligence Agency, 1970); but in these situations, all users are cleared to the highest security level of the messages in the system, and the system is not formally responsible for preventing unauthorized individuals from accessing stored messages of different classifications. The Sigma Message Handling System developed by ISI for the Military Message Experiment (Ames and Oestreicher, 1978) has been used to deal with different classifications of messages, although it was not actually secure. This experiment was significant in that Sigma simulated the behavior of one kind of system that might actually provide guarantees of separation among various levels of classification. But aside from these limited applications, there is little in the way of an experiential base to guide the designer of future message systems intended to operate in the multilevel secure environment.

Security is a pragmatic matter. There is a risk that something undesirable will happen to information stored in a computer, and that risk is balanced against the cost of various techniques and measures for protecting that information. Both the design and use of a secure computer system for a particular application must be based on an understanding of these tradeoffs. Hence, the purpose of this Note is threefold: (1) to expose some policy questions that affect security, functionality, and system efficiency; (2) to examine how security requirements in a multilevel secure computer system impinge upon message system requirements; and (3) to offer a taxonomy of options for merging the two sets of requirements, using current technology.

II. MESSAGE SYSTEMS OVERVIEW

The basic features of message systems are briefly reviewed here, to provide a basis for the more detailed discussions of technical and policy issues in subsequent sections. The NMCC Information Display Subsystem (NIDS) is outlined, as an exemplar of an important class of message system applications.

WHAT AUTOMATED MESSAGE SYSTEMS DO

A message handling system, manual or automated, must possess certain features to achieve a useful level of functionality. In manual systems, some of the features listed below are not explicitly identifiable, since they may reside in the user rather than the system. Other features, particularly those that implement support capabilities, are required only by automated systems (which may introduce new failure modes as well as new functions).

The users of automated message systems get some form of computer assistance in performing the following basic tasks:

1. Message receipt. New messages awaiting a user are brought to his attention and incorporated into his own files.
2. Message distribution. Messages are distributed to other users, either in the same system or on other systems on a common network; the user may forward (i.e., redistribute) old messages, in addition to sending new messages.
3. Message storage. Messages of more than passing interest can be saved for later reference.

4. Message retrieval. Saved messages can be recalled for further handling.
5. Message composition. New messages can be constructed, using new text, stored materials, or both.
6. Message editing and annotation. Stored messages can be modified by the user; a stored message may be an unfinished new message or an old message.
7. Message coordination. Several parties can coordinate and verify agreement on a draft message (this function and the message release function apply to systems operating under institutional constraints, e.g., in intelligence agencies).
8. Message release. The person responsible for the transmission of a message in the name of an organization can authorize distribution to specified recipients.

In addition to these basic functions, various support capabilities may enhance the functionality of an automated system; some may be required to alleviate new problems associated with an automated system. Fairly straightforward enhancements include

1. Report generators which provide the proper format for a variety of standard reports.
2. Statistics gathering subsystems which may, for example, keep tallies of messages by type, length, level, recipient, etc.

Support features mandated by the automated environment may include

1. Hold files for putting aside current work.
2. File backup and recovery procedures.

The principal motivations for the introduction of automated message handling systems are

1. Increased throughput, deriving from faster distribution, increased message composition efficiency, and more orderly and convenient filing.
2. Better accountability via system-supported audit trails.
3. Compatibility with other systems employing digitally encoded data.

SECURITY CONSIDERATIONS

It is unlikely that technologies will or should be developed that effectively replace prudence and personal responsibility on the part of users. It is reasonable, however, to seek assurances that the automated system is no more vulnerable to security violations than the manual system it replaces.

New potential risks may be introduced with the new capabilities. Some risks are associated with the increased volume of messages and the greater speed and ease of data transmission, both of which may amplify the effects of ordinary human error. To counter this, additional verification protocols might be imposed on the user, requiring him to corroborate his intentions before crucial actions are executed. Other risks are associated with the display medium, which may add anonymity and sameness to messages (e.g., the typical CRT display cannot provide

different type faces, colors, page sizes, etc.) and thus may reduce user sensitivity to the security level of current operations. This problem might be overcome by arranging to constantly display the security level at which the user is working. However, we must determine the potential severity of such risks before proposing to alleviate them through technological solutions.

In automated systems, just as in manual systems, apparent improvements in security are most easily made at the expense of user convenience and flexibility. Designers of secure message systems need to balance the potential benefits of decreased risk against the costs in user time, user compliance, and overall system function. Controlled and limited compromises to the system security model may be required to attain even minimum utility--for example, to allow the user to see a listing of all new messages. The potential risks inherent in these modifications must be evaluated in terms of exploitability by would-be penetrators, and not rejected out of hand because they violate the requirements of the basic security model.

THE NMCC INFORMATION DISPLAY SUBSYSTEM (NIDS)

NIDS is a CRT-terminal-based message handling system used by the NMCC for the dissemination of incoming messages to users, and for the subsequent manipulation, editing, filing, and redirecting of those messages. It also provides for the creation and internal dissemination of original messages.

Incoming messages are distributed to users who are logged on to previously defined positions (roughly equivalent to accounts), according

to interest profiles established for each position. Messages are placed in the message queue of each position whose profile criteria match that of the message. All incoming messages are stored in a message file, which always contains the most recent week's worth of messages. The user may retrieve copies of messages from the message file into his pending action queue by submitting a search query to the system. The items in the message queue may be scanned by displaying the first page (screenful) of each message. They remain in the message queue until they are explicitly deleted. Messages and user-created text files may be edited and otherwise manipulated, stored in the work file belonging to the user's position (indexed by user-defined categories), and/or sent to other positions on the system. The system also allows for position-to-position conversational communication.

In addition to the above, there are special system support procedures for watch change (i.e., logging on/off a position), profile construction, message file search-query specification, report generation, file printing, short-term file storage, and user status reporting.

NIDS makes use of the following set of files and queues:

1. Message queue. A set of messages ordered by precedence set by the originator, available to a user logged on to one or more positions. The messages may be scanned sequentially forward ("next") or backward ("previous"), or the user may return to the first message. As each message is accessed, the first screenful is displayed and the user may manipulate the entire message--he may delete it from the message queue, save it in one of several files, send it to other users, edit it, etc.

2. Message file. A collection of messages entered into the system during the previous six days, with the oldest messages being purged as additional file space is required. Users may retrieve copies of messages by submitting search forms specifying Boolean search criteria on subject matter and other variables.

3. Work file. A file associated with each position, into which messages and other items may be stored by user-defined subject category. Items may be stored under more than one category. For each position, NIDS maintains an index of subjects and the date-time group of storage of each item. A user may both store into (write) and retrieve from (read) work files associated with the position(s) into which he is logged. A user may only retrieve copies of items from work files belonging to other positions. There is also a special work file (called OMNI) for which all users have both store and retrieve privileges. Items may be retrieved from work files in several ways:

   a. An entire set of items within a subject category may be retrieved and scanned for items of interest.
   b. Items may be retrieved by specifying the storage date-time group.
   c. Items or sets of items may be retrieved by specifying unions/intersections of subject categories.

Individual items may be scanned page by page, and an entire set may be scanned sequentially forward or backward. The item currently displayed is treated as a single unit and may be printed, deleted from the list of response items, etc. Items may also be replaced in the work file after editing, or may be deleted from it.

4. Hold queue. Each position has an associated hold queue into which current items may be placed when more urgent tasks must be performed. The hold queue is sequential and may be scanned forward or backward; it is also possible to jump back to the first queue entry. Pages of an item in the hold queue may be scanned sequentially. Items remain in the queue until they are explicitly deleted.
5. Pending action queue. Each position has a pending action queue that receives intra-NIDS communications as well as retrievals from searches of the message file.
6. A file of standard report formats from which the desired format may be chosen, filled out, and then printed, filed (in the hold queue or work file), and/or sent to other positions.

A variety of system functions permit the user to manipulate the contents of the several files and queues. In functional terms, the user may

1. Log on/off of one or more positions (watch change).
2. Build, store, and modify message search profiles associated with each owned position.

3. Scan the message queue and select items of interest.
4. Edit received messages and create original messages and reports.
5. File/retrieve received messages or original documents in an index-structured work file.
6. File/retrieve current items in the hold queue.
7. Retrieve copies of recent NIDS messages by constructing and submitting search queries.
8. Communicate with other positions either conversationally or by message.

Conceptually, the user "owns" a copy of any item (message, report, etc.) that is the current item displayed on the terminal. If the user modifies a retrieved item and wishes to preserve the changes, he must explicitly save it (in a work file or hold queue), print it, or send it.

The user accesses the system through a dual-screen terminal comprising a typewriter-like keyboard, special function keys, and a light pen. A portion of one of the screens is generally reserved for the entry and display of commands, function menus, system messages, and error messages. The balance of that screen and the entire second screen are used for text display and editing.

The user may change activities at any time by pressing a special function key (ENTER COMMAND) and then either typing a command, pressing another function key, or selecting an item with the light pen. The desired activity does not take place until another special function key (EXECUTE COMMAND) is depressed.

The function keys that are appropriate to a particular context are lighted when in that context, and a list of possible commands is also displayed. Certain function keys (e.g., PAGE FORWARD) are utilized in several contexts, while others (e.g., PUT IN HOLD QUEUE) are single-purpose.

III. SECURITY POLICY AND COMPUTING SYSTEMS

The security of sensitive information is affected not only by physical and technological conditions, but also by legal, administrative, psychological, institutional, and other considerations. At root, security depends on trust, which at times may be misplaced--whether in a machine, a person, a procedure, or an organization. It is the interplay of all the various human and technical factors that determines the effective degree of security in any given situation.

Decisions concerning security methods will often involve complicated tradeoffs among the risk of disclosure of important information; the utility of the information (to both friend and foe); the ability of the authorized users to manipulate data conveniently; system costs for protection mechanisms (measured in dollars, time, performance penalties, loss of functionality); and a host of other factors. The following five technical issues interact closely with policy decisions in affecting the ways secure message systems (and other applications in secure environments) are designed, built, and used:

1. Granularity. The size of the smallest data unit that can carry its own security label within the system.
2. Trust. The level of confidence that can be placed in various elements of a system handling classified materials.
3. Confinement. The ways in which computer processes can be prevented from using indirect means for leaking protected information.
4. User interface. The impact of system security features on user perceptions and behavior.

5. Integration. Operation of secure message systems in joint environments with other related applications software.

GRANULARITY

Granularity refers to the size and status of the smallest objects that can be individually protected within a computer system. The objects of interest here are such things as files (at the system level) and messages (at the application level). There is a good deal of variation among computing systems in the way these objects are stored and processed.

A file may contain a single user document, or in some computer systems, files may contain several documents--or logically distinct blocks of text--physically packed together and referenced by a single system file name or pointer. From the user's point of view, such a file might constitute an entire database or a single report. Conversely, a user's database might be stored in one file, or in several files, depending upon the host system.

But a database itself may have a great deal of internal logical structure, and parts of the database may be more sensitive than other parts. Even at the level of a single document, different paragraphs of a classified report often have different security classifications. Some systems that store messages pack the individual messages together into a small set of files, while in other systems (Borden, Gaines, and Shapiro, 1979), each message is an individual file. Most sophisticated computer systems further allow the aggregation of files into directories. Obviously, serious problems can arise in attempting to harmonize the

user's concept of a message, the system's concept of a file, and the security policy's concept of the appropriate minimal unit for individual protection.

Let us consider this in somewhat more detail for the case of military messages. The top-level structure of a message divides the message into a "header" and a "body." The body is the text of the message, while the header contains a variety of information about the message, the sender, the set of recipients, the date and time of sending and receipt, the precedence of the message, its classification, etc. Some systems include a good deal of additional information in the header as well. The body of the message can be anything from a single line to a lengthy report. The header and the body may be of different classifications, and individual elements within the header and the body may themselves carry different classifications. For example, within the header, the subject may be more highly classified than the other elements. The header is frequently either unclassified or has a low classification, while the body may be highly classified. Finally, different paragraphs within the body of the message may have different security classifications.

Now the question arises, How should the problem of mixed classifications be handled within a computer system? One approach is to store each individually classified element of a message (or other object) as a separate file within the system. But if this is done, a potentially complicated indexing structure is needed to represent the entire message as a unit within the system.

Alternatively, messages could be stored in a pair of files, one containing the bodies of messages and the other containing the headers. This would allow different classification of headers and bodies, while maintaining the convenience of security granularity at the file level (i.e., a single level of classification for any given file). In this scheme, the bodies of all messages would carry the same security classification, and the headers likewise would all carry a single classification; for example, all the bodies might be Secret and all the headers Unclassified. An important advantage of this arrangement is that certain useful programs which require access only to the headers would only need access to an Unclassified file. Thus, it would be possible to build a program that produces a summary of the messages in a folder, showing the originator, the subject, and the date and time the message was sent, etc., without requiring access to classified files.

But this simplification of the software may be purchased at the expense of added problems for the user. A user who wishes to annotate the header while examining a message body may find that he is not operating at the appropriate security level. According to the basic, widely used security model, anything written by a computer program must be classified at least as highly as the program itself. Thus, if the user were operating at the Secret level to read the body of a message, his Secret-level program could not modify the header information stored in an Unclassified file.

Another approach would be to store each message (header and body together) in one file, which might contain only one message or several packed messages. Again, this assumes that the security granularity of

the system is at the level of the file. This simplifies the message handling software, because the program that displays the message needs to read only a single file to obtain the entire message. If the user wishes to modify the message (by making annotations, for example), he can make appropriate markings in both the header and the body. However, this approach entails the "artificial" upgrading of Unclassified headers to the Secret level, so that they can be stored with the Secret message bodies. Downgrading will then be required to make the headers available to lower-level processes. Of course, many alternatives for dealing with this problem are available to the designer, but no matter which approach is favored, this set of issues must be confronted.

The level of security granularity affects the complexity of the system. As we protect smaller and smaller objects, the system must handle larger and larger numbers of them. As the numbers increase, the size and complexity of the supporting system (indexes, directories, pointers, etc.) also increase. If user-level objects normally consist of many pieces, each at a different security level, mechanisms for dealing with these pieces as a whole will be needed in addition to mechanisms that allow us to deal with the individual pieces. The system must be able to efficiently access a number of small, scattered pieces in quick succession if the granularity is very fine. On the other hand, if the granularity is too coarse, the system and user may be forced to operate at a single, high security level, when in fact, pieces of the object are of lower classification. This raises the problem of downgrading--removing information of a lower classification from an object of higher classification.

The latter case superficially resembles the situation of a person dealing with written documents. A document is marked with the classification of the most highly classified information it contains; pieces of that document may have lower classifications. But in this case, downgrading is not a problem. The person who wishes to copy an unclassified paragraph from a classified document may do so because he has been trusted with the entire document in the first place. However, a computer program that does the same thing raises concerns about the correctness of the program, in addition to the intentions of the user when he uses that program. It is this problem of guaranteeing the correctness of the program, or arriving at another reason for trusting the program, that makes this issue difficult. We will return to the question of trust below.

The user generally finds it simpler to use a system if large objects having some internal logical coherence can be treated as a single unit. For many purposes, it is simpler to manipulate a large text file as if it were all of one classification, even though some parts of it may be at a lower level. For example, it is easier to copy large sections of the file into another report of the same classification. It is also often preferable to treat a message consisting of a header and a body as a single unit; this is the appropriate way to forward a message, for example. A fine-grained secure system might entail frequent, cumbersome changes of security level. If the user must make such changes frequently while dealing with what he normally thinks of as a single object, he may seek ways to defeat the system in order to avoid the annoyance of what he perceives as wasted effort.

Here, in a nutshell, is the granularity tradeoff: For many purposes, coarse granularity leads to simpler design in the supporting system and in application programs, and to greater user convenience. However, it is generally more difficult to abstract information of lower classification from an object of higher classification in a coarse-grained system. And it may be difficult to provide supporting software that can be trusted to help with this task. These points will be elaborated below, and examples of this tradeoff will arise in the discussion of particular approaches to message system design.

TRUST

At the root of any security system is some sort of trust. The trust can be based on confidence in an individual, validity based on inference, knowledge of the context in which something was designed or is being used, and a number of other factors. Computer programs are notoriously difficult to make error-free. In addition, computer operating systems, which are very large computer programs, have been shown to have many security flaws even when unusual precautions have been taken in their design and implementation. In recent years considerable effort has been devoted to the development of techniques by which the security properties of computer operating systems can be formally specified and verified. This has been a major motivation for the growing area of computer program verification. But formal verification methods cannot be depended upon to answer all issues of computer security (DeMillo et al., 1979).

One of the major obstacles is complexity; it is especially difficult to verify large computer programs. As a result, a substantial effort has been under way for the last several years to produce operating systems in which a "kernel," or a small part of the operating system upon which the rest depends, can be isolated and formally verified. This is the approach taken with the KSOS (McCauley et al., 1979) operating system--a system which functionally resembles UNIX, but which was built from scratch to be secure. Even with development of automated verification systems, verification is a reliable source of trust for only a small fraction of the code that is really relevant to security in a large computer system. This implies the need to justify confidence in a collection of programs that comprise the computer system being used, even when all those programs cannot be verified.

One basis for confidence in a computer program is confidence in the person who wrote it. But for certain kinds of code, this won't suffice. The problem is not that the designer or coder of the program is untrustworthy from a security point of view; it is simply that the possibility of error is so prevalent in computer software that one cannot rule out inadvertent security flaws. For operating systems that must operate absolutely correctly in order to guarantee appropriate access protections for stored data, simply trusting the good intentions of the programmer will not do.

Under other circumstances, this kind of people-based trust may be quite sufficient. For example, a text editor must access the files in the system that are to be edited and provide a set of editing functions that allow the user to manipulate the text. For simplicity, we may

assume that the interactions of the text editor with the system consist mainly of opening a file, using the data read from that file as a basis for generating a new file during an editing session, and writing that new file when the editing session terminates. In addition, during the course of editing, the text editor must transfer data to a terminal to be displayed to the user and receive the keystrokes that the user types as he proceeds with his editing. These interactions with the operating system are quite straightforward.

Editing programs, like other complex programs, can be expected to have bugs in them. However, most of these bugs will affect text editing performance but not security-related performance. For a text editor to make an error from a security point of view (with one exception discussed below), it must somehow gain access to a file that the user is not authorized to access. But the underlying operating system is designed and certified to prevent this. Hence, any failure on the part of the editor to perform correctly will be a performance failure but not a security failure.

The one exception to this is the case where security markings are present in the file being edited--a situation that can arise when the security granule, in systems terms, is finer than the file. Typically, in classified documents, each paragraph carries a symbol indicating the security sensitivity of that paragraph--a (U), (C), (S), or (TS) at the beginning of the paragraph indicates that the paragraph is Unclassified, Confidential, Secret, or Top Secret. A security error can occur in the text editor if these security designations are deleted or modified. If we do not trust the programmer who writes the text editor, we might

suspect him of deliberately constructing the editor so that it changes some of the security markings. For instance, it might occasionally change a (TS) to a (U). Then, at a later time, highly classified paragraphs could be moved into an Unclassified file because the internal designation was no longer correct and misled either a program or a human being who used the file.

It is highly unlikely that this kind of change could occur by accident. An accidental modification of a line of text by a text editor is unlikely to produce an exploitable result. The security designation and other parts of the line may be deleted or modified in some way that is not helpful, but the particular modification that changes the legal security label into another, lower, legal security label is very unlikely to be an accidental error.

An important aspect of security is detection. We not only try to prevent unauthorized violations of security by means of access controls on the use of files, we also count heavily on surveillance and detection to deter potential security violators. The performance of a text editor is highly observable. Every user can observe what the text editor does to the files that he edits. Therefore, deliberate program-based modification of such designations must be done in such a way as to escape detection by the user. But the output of text editors is inspected in great detail, and it is highly unlikely that even an occasional modification of a security designation could go undetected for long. Thus, in addition to the trustworthiness of the programmer, we can count on the deterrent value of highly probable detection for assurance that use of a text editor will not compromise security.

This suggests that we can justify a higher degree of confidence in the security properties of at least some applications programs, without the more rigorous (and costly) approach of formal verification. As a policy matter, it is important to distinguish applications where expensive methods such as formal verification are required to achieve an adequate level of security from those where such assurance can be achieved by less expensive methods.

CONFINEMENT

The "confinement problem" (Lampson, 1973; Lipner, 1975) refers to a class of security flaws in computer systems that may pose special problems for multilevel secure message systems. These flaws involve the leakage of information through indirect channels we define here by exclusion: They are not the normal ways by which programs output information. Rather than writing information into a file or to a terminal, the program causes changes in some variable which is observable by others, but which is not intended to be a communications path. For example, a subversive program may create empty files in a directory so that someone (or perhaps another subversive program) can later check to see how many empty files there are. Thus, this becomes a signaling path by which a program could communicate illicitly to someone whom the security mechanisms of the system would prevent from reading files written in the normal manner. This problem is potentially serious; an experiment was conducted in which a terminal was driven at a high data rate by a program that watched a directory as another program manipulated it. Lampson discusses the technical aspects of this problem

in some detail. Here, we focus on the policy considerations it involves. As will be seen, much of the above discussion of trust is applicable here.

Communication requires a sender, a receiver, and a channel. Those who worry about confinement have identified a large number of potential communication channels in computer systems. These include such system data as the number of files in a directory, the length of a file, etc. Effort and coordination are required on the part of both the sender and the receiver to use such a channel. Furthermore, for most confinement channels, the sender and receiver (or their programs) must be active at the same time, and both must engage in fairly substantial activities to exploit the potential confinement flaws. Finally, while the data rate of some of these confinement channels may be fairly high under rare and special circumstances, the rate of change of the signaling variables may be quite low during normal use of the system.

These factors may permit ample opportunity to observe either the sender or the receiver in guilty behavior. In contrast, for many operating system penetrations, there is little or no opportunity to observer the penetrator; he can often do his work in preparation for the penetration entirely unobserved and then run a program that is active for only a brief instant within the system to actually accomplish the penetration. The use of confinement channels exposes the perpetrators to a far more significant risk of detection.

A program that exploits a confinement channel can only have been written by someone who wishes to defeat the system. Thus, one rather direct way to avoid this problem is to trust only programs written by

people who can be trusted. This means that sensitive information cannot be handled by utility programs whose source is unknown or unverified. This policy may eliminate a lot of useful applications software or cause it to be rewritten, but it is clearly less expensive than formal verification of such code would be. Short of employing trusted people to produce all code that will handle sensitive information, it may be deemed sufficient to carefully examine such code. The kinds of system variables that can be observed are not particularly easy to manipulate and may not easily escape notice if a program is rigorously scrutinized. One might worry about accidental confinement violations that a would-be penetrator might discover, such as the accidental but noticeable change of system variables which correlates with the contents of one or more protected files. It is difficult to imagine exploitable flaws of this type; such situations seem highly unlikely to arise in practice.

What is the likelihood that a program will be deliberately designed to communicate by violating confinement? Such a program would probably require a fairly large amount of computing time to achieve a useful data rate over a confinement channel (except in the case where a very small amount of leaked data would constitute a serious security problem--probably a very rare circumstance). The program must do this in a way that escapes notice. But program running time is often a matter of concern, and the overhead for such actions as creating and deleting many files is substantial in most systems as well. Thus, this aspect may make a program that violates confinement relatively open to detection, with the consequent risk of detection of the person who supplied the program.

This Note introduces a number of different design possibilities for message systems. The ones that offer the greatest convenience to the user or that are technologically least costly to implement generally appear to offer the most plausible opportunities for a penetrator to exploit the confinement channel. But a careful assessment of the actual risks that would be incurred should be carried out before these otherwise more attractive approaches are rejected. Little work has yet been done on the feasibility of monitoring for confinement violations, beyond simply trying to prevent the channels from existing in the first place. It is not out of the question that for many known confinement channels, reasonable monitors will be much cheaper than actually blocking the channel, and may lead to systems as secure as they would be if no such channel were present. The question of monitoring confinement channels is not examined in this study, but it merits attention.

USER INTERFACE AND CAPABILITIES

We now consider the kind of user interface that is desirable for a multilevel secure message system. Should the user be continuously informed of the security level of the data he is dealing with? When he attempts to access a piece of data that he has been allowed to see before and that is at a security level different from that of the current data, should he be asked to verify that he has requested a change in security level?

Such questions, and others concerning the appearance of a multilevel secure system to the user, involve a tradeoff between convenience and utility on the one hand, and the need to insure that classified material is dealt with in a secure fashion on the other.

Today's computers are not smart about people's intentions and cannot generally recognize whether a particular act is valid, if the same act could sometimes be a violation of security. People have the ability to mix information of different classifications in their heads and so can have access to high-level material when, for example, they are preparing a document of low security classification. But the policy rule that forbids programs to write files of lower classification than the highest level to which they have access is fundamental to the design of secure systems (Ware, 1979). Should the writing of lower-level files be permitted when the user verifies that it is intentional, on the grounds that the user was trusted if he was given access to data of higher classification in the first place? To what degree should such copying of data across security levels be made convenient, and how likely is it that such convenience would lead to accidental violations of security? To a large extent, such questions can be answered only through experience gained from the actual use of multilevel secure systems.

The user's interface to security downgrading capabilities may constitute a major policy problem. Today, systems are being explicitly designed to prevent downgrading, so that they will be safe from exploitation. But a downgrading capability is absolutely necessary at times. The difficulty is in knowing when it should be permitted and how it should be controlled. It has been suggested that every piece of information copied to lower levels should be displayed on a terminal for the user's review, so that he can verify and acknowledge the correctness

of the downgrading. But in many situations this could be onerous indeed and might undermine the utility of the system. The appropriate answer is not yet clear, and it probably will vary from situation to situation. Some specific cases that arise in the design of secure message systems will be mentioned subsequently.

INTEGRATION

Designers of secure message systems need to remember that a message system is only one of several user tools and that it must be skillfully integrated with other capabilities. As computer use becomes increasingly interactive, both the source and the destination of much of the information that is transmitted by messages will lie outside the message system. In a tactical command and control environment, for example, much of the information in messages relates to forces used in combat; the data often come from one database and are used to modify or update another database. If the user is forced to copy information manually from a message because it cannot be used directly to update information stored in the computer, the system is not assisting him to the degree that it could. If the basic capabilities of the message system itself are not integrated with convenient means for manipulating text and other data (database management, word processing, etc.), the user will often not obtain full advantage from the system.

Message systems were initially designed almost exclusively to send messages from one person to another. This explains why so many message systems have been built with little or no concern for integration with the rest of the computational capabilities that may exist in the host

systems. Certainly, the rapid delivery of text messages, typed by hand and intended only to be read by humans, is an extremely important capability. The speed and convenience provided by computer-based message systems are substantial advantages, even when the systems are used only in the human-to-human mode. But the utility of these systems is far greater when they are integrated with other automated information processing functions.

Information received by message can be expected to be used for briefings, for updating intelligence and logistics data, for assisting in management and direction of operations, and for planning. It must therefore be possible to quickly move relevant information from messages into other forms for computer processing. Similarly, it will be increasingly necessary to obtain information for inclusion in messages from files of various kinds within a computer system. The integration of message systems with other computer information processing will take time and effort, but it cannot and should not be avoided. The development of stand-alone message systems that are disjoint from other systems or totally segregated within a single larger system is anachronistic and counterproductive.

The integration of message systems with other systems has security implications. The availability of a convenient method for sending out messages containing information from files increases the risk that data stored in a computer system will go where they should not go. The security policies for the message system and the means of implementing them must be consistent with those governing the rest of the system. It will do relatively little good to segregate paragraphs of a message by

security level if the processing of that message by other software in the system requires the entire message to be treated at a single level of classification. Hence, the designer of a multilevel secure message system must take into account the security capabilities and requirements of both the host operating system and the companion applications that will run on that system.

IV. APPROACHES TO THE IMPLEMENTATION OF SECURE MESSAGE SYSTEMS

This section presents a number of different approaches to the design of message systems in multilevel secure host environments. Each approach has its characteristic strengths and weaknesses. First, we outline a simple taxonomy of system properties that characterize and distinguish the various approaches. Next, we present a set of generalized (and quite familiar) evaluation criteria by which these strengths and weaknesses can be loosely compared. The choices in actual implementations will depend upon the missions and resources of the intended user community.

A TAXONOMY OF DESIGN APPROACHES

The design approaches outlined in this section differ from one another along two dimensions:

1. The degree to which the system supports user-level objects of mixed security classification.
2. The degree to which the system assists the user in overcoming the inconvenience of the strict system-level separation of materials at different levels of security classification.

Along the first dimension, four classes of systems are considered:

1. Complete isolation of security levels: single-level messages; single-level folders; message processing at only a single security level at any one time.
2. Multilevel folders of single-level messages, enabling the processing of groups of messages at various levels at one time

(e.g., sequentially reading and deleting messages of different security classification levels).

3. Separation of messages into header and body, at different security levels.
4. Multilevel messages, marked by paragraph or word.

Along the second dimension, six distinct technical design approaches are reviewed:

1. Strict adherence to the secure host's security model, with no additions to the basic message system and no modifications to the host operating system. All processing is done within the security model.
2. Manual upgrade and downgrade. The user upgrades and downgrades messages at his discretion, and system security relies on his attention and judgment.
3. Upgrade with protected markings, automatic downgrade. A special trusted mechanism puts its own markings on data and downgrades automatically.
4. Protected context file with pointers to data at multiple levels. The data themselves are protected by KSOS; the pointers allow process and control information to flow between levels.
5. Secure message system server (similar to the KSOS secure server). The server switches processes automatically, depending upon the security level of the data being accessed.

6. Trusted code. The entire message system is built with trusted code and can violate the host system's security model in any way in the pursuit of functionality.

Figure 1 presents a convenient guide to the subsequent discussion of message system designs based on this taxonomy. A number of key design features are incorporated in the various systems for enhanced functionality and convenience. These features are described here as they would be implemented in UNIX and KSOS, but similar features could be implemented in any multilevel secure system.

The first feature is a mechanism to simplify changing security levels. In KSOS the user changes levels by asking the secure server to switch his terminal to a different process of the desired level (an individual process cannot change its own level). Thus, all of our designs require a mechanism for changing to another level with a minimum of user effort. Many times, the level that the user wants can be straightforwardly deduced by the previous process. This suggests a mechanism by which a process might store a target-level specification in a buffer in the secure server; the user could change to the new level by simply confirming it. Such a feature would be most useful in situations where the user must change levels often to continue processing on the next logical data item. A process switching feature adds only a minimal cost to the server (the cost of verifying the additional mechanism). It does not violate the security model and thus creates no additional security problems.

Granularity	Strict Security Model	Manual Upgrade and Downgrade	Manual Upgrade, Automatic Downgrade	Protected Context File	Secure Server	All Trusted Code
Strict Single Level	Design 1.1 (p. 42)	Not Applicable	Not Applicable	Not Applicable	Not Applicable	Not Applicable
Multilevel Folders	Not Applicable	Design 2.2 (p. 46)	Design 2.3 (p. 48)	Design 2.4 (p. 51)	Design 2.5 (p. 55)	Design 2.6 (p. 56)
Separated Multilevel Header and Body	Not Applicable	Design 3.2 (p. 58)	Design 3.3 (p. 58)	Design 3.4 (p. 59)	Design 3.5 (p. 60)	Design 3.6 (p. 60)
Multilevel, by Paragraph or Word	Not Applicable	Design 4.2 (p. 63)	Design 4.3 (p. 63)	Design 4.4 (p. 64)	Design 4.5 (p. 65)	Design 4.6 (p. 66)

Fig. 1--Taxonomy of multilevel secure message system designs

Several of the designs presented below use the read-up property of the security model to allow a user to peruse messages of lower levels from a high-level process. A slight deviation from the security model would allow a useful adjunct to this capability: the ability to delete lower-level messages when a user has finished reading them. The DEL ("delete file") command has a limited potential for this. It requires a limited confinement channel that allows the user to delete files of any lower level while the bandwidth of the DEL command is limited in various standard ways with a monitor. For instance, a specialized DEL for removing messages could be constructed. This command would be invoked by a process taking its input from a terminal, would have a time delay to limit its bandwidth, and would be protected against modification. At run time, the specialized DEL command would run more slowly, but it could be superseded by a normal DEL command in cases that do not violate the security model. The necessary monitored confinement channel, which would have to be evaluated as a potential security risk, and the development of the confinement channel monitor would, of course, add to the costs of the system.

Many of the proposed designs include a downgrading program--a piece of trusted software that allows a user to downgrade information from one file to another and assume responsibility for the downgrading. The downgrader in turn reports to the security officer (through an audit trail) what information was downgraded, when, and by whom. The common assumptions about such a facility are

1. It will not be used so often that it will swamp the security officer with audit information.

2. The user will have to interactively confirm each piece of information to be downgraded (he will have to be presented with each paragraph and be required to execute a keyboard confirmation) to reduce chances of unintentional downgrading.
3. The downgrading operation will take a sufficient amount of the user's time to discourage its casual use.

The downgrader itself implies no additional security risk to the system. A malicious user can always write the information down on a piece of paper and retype or copy it as he wishes.

Finally, some of the designs rely upon creating secure subsystems within the secure operating system. A secure subsystem is defined as a set of programs and data files that have their own built-in security checks and enforce their own security policy, above and independent of the operating system's security policy. Of course, the subsystem's security must rely upon the host operating system's security. When the subsystem writes a file that it wishes to protect, the operating system must recognize the subsystem's access lock and respect it. A typical technique is to create a set of objects (e.g., files) that can be accessed only by a particular set of programs. These programs must then be trusted. Two special trusted programs must be used to enter new objects into the subsystem and to release objects back to the operating system's protection.

KSOS provides a method for creating such secure subsystems, through its subtype manager. Files can be designated to be of a special type called a subtype; only processes having a special subtype capability can access them. Normal processes can invoke protected processes, but these

processes are then protected from further manipulation by the calling process. The invoked subtype process can then access the subtype-protected files.

With such a technique, a system designer can create an entire secure subsystem within the host secure operating system. This subsystem can then be used to implement limited confinement channels or special security policies that violate the security model or its KSOS implementation. For example, a set of normal files of several security levels can be mapped into a single file in a special representation by a secure subsystem process and stored as a single file in the operating system. This file can then be read by untrusted applications software. When the file is to be modified, it can be remapped into separate files and removed from the secure subsystem. Security in the subsystem then relies solely upon the two mapping mechanisms.

EVALUATION CRITERIA

Several kinds of value must be considered in assessing the fitness of different designs for particular message system applications in multilevel secure environments. In actual practice, the criteria listed below will need to be supplemented by others related to the particular circumstances of individual cases. The lack of common measures among the different sets of criteria may require difficult tradeoffs in making decisions about system procurement. The basic evaluation criteria include

1. Functionality, or the range and convenience of the functions provided by the system. All of the design approaches discussed

below provide the basic functions required in useful message systems (reading, composing, saving messages, etc.). Over and above these, some designs incorporate additional important conveniences, such as

a. The ability to easily process streams and collections of messages of mixed classification.

b. Automated assistance in message downgrading.

c. Convenient whole-message operations (e.g., forwarding, deletion, filing) on mixed-level messages.

2. Ease of use (human-engineering factors). For multilevel secure environments, two particular factors emerge:

   a. How much additional typing/interaction must the user do (over that required on an equivalent system with single-level security)?

   b. Is the user likely to understand exactly what his security responsibilities are at all times?

3. Level of trust, or how characteristics of the design affect the cost and feasibility of code verification (and/or other measures) needed to certify the systems as trustworthy or secure:

   a. Is there a standard certification method, e.g., verifying the design and implementing code?

   b. Is the design simple, e.g., are all of the security violations contained in a partitioned "kernel" of system calls?

c. How grave are the threats to which the system is potentially vulnerable?

d. What is the potential cost of a security breach?

4. Cost/time/efficiency. There are several factors that are unique to multilevel secure environments:

a. Software development cost and time. Verified or otherwise secure-certified software takes significantly longer to develop and costs more.

b. Secure storage cost. Highly classified data require more stringent and therefore costlier security procedures for storage device management. Thus, it is important to minimize the security level of stored data.

c. Scaling up. Designs that are viable for small data files and limited numbers of users must be tested for viability in large applications.

d. Storage cost vs. time cost. Most designs trade off excessive process switching against more complex data structures.

OVERVIEW OF DESIGN ALTERNATIVES

We shall now describe several feasible approaches to the implementation of message systems in multilevel secure host environments. These approaches are based upon the taxonomy of methods and features described at the beginning of this section, and they are evaluated against the criteria outlined above.

1. Strict Isolation of Security Levels

In the first category of design alternatives, processing of data at one security level is completely isolated from processing at all other levels. Thus, in message systems, all messages have a single security level, all folders have a single security level, and only messages of a single security level may be processed at any one time. This represents the default (least-effort) approach for message systems in a KSOS-like environment, since the message system applications software strictly conforms to the security model implemented in the host environment: Processes may read and write data only at their own security level.

> Design 1.1*: Untrusted software that strictly conforms to the security model of the host operating system.

This category of designs is exemplified by the Secure Message Handler (SMH), developed at Rand to run on the KSOS operating system. SMH is an adaptation of MH, a system developed at Rand to run under UNIX, taking full advantage of the flexibility of the UNIX file and directory system.

MH has three definitive design characteristics:

1. Each message is stored as a UNIX file, one file per message.
2. Each folder is stored as a UNIX subdirectory of files.
3. Each MH command is a separate program invoked via the UNIX command interpreter (shell). Thus, MH commands can be

*Design numbers reflect the class and design approach in the taxonomy given on pp. 33-35. Thus, Design 1.1 represents system class 1 and technical design approach 1.

invoked in any system-level context and can be used as filters in pipelines with other programs. A dedicated context file and a set of user profiles keep track of the relevant context from one command to the next.

MH has a flexible set of fairly standard message handling commands to scan message headers; display messages; delete messages; compose messages; create, delete, and access folders and file messages in them; and select messages based on their contents, origin, date, etc. Each MH command runs a separate program, potentially reading and updating the context file.

The major difference between SMH and MH is the separation of message storage by security level. In SMH, each set of messages and folders of a particular level is stored in a separate directory of that level. Whenever the user wants to process messages of a given level, he must start up a process at that level, via the KSOS secure server. Each SMH command, on invocation, notices the security level, reads the context file for that level (which includes pointers to a directory of all folders of that level), and processes messages only from that level. One typical consequence of the strict separation of levels is that the user can fetch incoming mail from only one level, thus potentially receiving only a subset of his incoming mail; later retrievals at a different level may therefore include messages that actually arrived at the site earlier.

SMH embodies two features that are anomalous with respect to the KSOS security model and that require secure code in the application domain:

1. Notification of all incoming mail, regardless of security level, at the time it is received by the system.
2. A downgrading facility for moving overclassified materials from one message to another, lower-level message or file.

Functionality. Obviously, multilevel messages are impossible to represent in a system of this type. Messages containing data at different levels must be classified with the highest level of data they contain. Folders with messages of multiple levels cannot be accommodated. Finally, users must utilize the downgrader to move low-sensitivity data from a message that also contains high-sensitivity data to a lower-level file.

Ease of Use. Design 1.1 requires either that all messages on a particular topic be at the same security level, or that the user personally keep track of topic relations across levels, e.g., with mnemonics. Users may in fact tend to upgrade related messages to the level of the highest-level messages in the set, thus substantially raising the burden of subsequent downgrading. The security model, however, and the user's responsibilities for working within it are completely clear and understandable. User interaction cost will tend to be high, depending upon how much level-switching and bookkeeping between levels is required.

Trust. Except for the anomalies mentioned, trust is system-high, i.e., as high as the underlying operating system that enforces the security model. All other applications software can be totally untrusted.

Cost. Design 1.1 entails minimal additional cost for applications software, mostly for extra system code to retrieve and keep track of the current security level, and for the downgrader.

For situations where strict separation of security levels is to be maintained, this design is quite adequate, almost by definition.

2. Multilevel Processing of Single-Level Messages

The second category of designs takes advantage of a limited mixing of security levels for greater functionality and ease of use. Here, it is assumed that each message has a single security classification which applies to the entire message; in limited ways, messages of different levels may be grouped for storage and processing. The category includes five design alternatives, each of which allows multilevel folders of single-level messages and manipulates sets of messages at various levels without switching process levels (e.g., reading or deleting messages of different levels from the same process). The most important functional benefit is that the user can easily process streams of messages (e.g., incoming mail, or sequences of messages in a folder).

It is explicitly assumed that the operating system security capabilities will be used to protect each data file at its particular level (i.e., Secret messages will be stored in files whose security level is Secret, etc.). A multilevel folder must somehow "contain" (point to) these messages and thus allow the user to associate a logical name with the message collection. The problem, then, is to construct a message system in which the user can read and write messages stored at

various levels in these collections. In KSOS, one implementation of a multilevel folder would entail a set of subdirectories of a main folder directory. The subdirectories would each have a different security level and would contain all and only the messages in the folder at that level. Special secure software is necessary to move folders, because directories of different levels must be read and rewritten.

Design 2.2: Manual upgrade and downgrade.

The basic strategy in this design is to manually upgrade all information to be processed to the highest level of any included subset, process it, and then manually downgrade the pieces back to appropriate levels. Both upgrading and downgrading are under the user's control and responsibility. System security relies upon his attention and judgment.

Specifically, this design entails upgrading all messages in a particular folder to the security level of the most highly classified included message (in practice, often the highest level that the user may plan to access), processing the messages, and downgrading any remaining messages to their appropriate levels.

The upgrading program must tag the messages with their original level or must provide some other form of bookkeeping to aid the user in reclassifying the messages. This bookkeeping aid will not be protected in any way; thus, malicious programs could rewrite information at this level. The user is responsible for downgrading and checking to make sure the data are reclassified to the correct level. Some additional processing at each level may be required to return the messages to their proper file locations (e.g., their original folders). This design

requires no trusted processes (beyond the downgrader) and is probably most viable for situations in which the processing consists of reading new mail and most messages are discarded rather than being reclassified.

Functionality. Design 2.2 gives the user enhanced functionality over Design 1.1 if the convenience afforded the user by performing operations on a set of messages of differing security levels is greater than the effort required to downgrade and relocate messages to their proper levels. This design does allow message system operations on sets of messages of differing intrinsic security levels. However, the user cannot in general manipulate the messages with other applications software before reclassifying them, since he must keep track of the necessary security markings.

Ease of Use. The user must constantly upgrade and downgrade. If the downgrader is strictly manual, as described above, this could be very cumbersome. The user's security responsibilities are relatively easy to understand, but they are great and frequent.

Trust. This design puts a heavy responsibility on the user to downgrade messages correctly. The messages may be varied and presented in a random order, potentially causing confusion. This is a situation in which the "anonymity" of the CRT display could be detrimental, from a security perspective.

Cost. In operation, many files will be upgraded to the highest security level. If the security procedures for file storage of this level are costly and the inconvenience of manual downgrading encourages widespread overclassification, the excess high-level files may add significantly to the system operation cost.

Design 2.3: Upgrade with protected markings, automatic downgrade.

The strategy in this design is similar to that for Design 2.2; the user upgrades, processes, and then downgrades a particular set of messages. The difference is that in Design 2.3 a secure subsystem is responsible for protecting the markings of the original level. Because the markings are protected, the secure subsystem can also take responsibility for downgrading. The secure subsystem itself would be protected by the operating system security system; this is the function performed by the subtype manager in KSOS.

This system is used in the following way. The user invokes a special upgrader from the highest security level he needs. The upgrader upgrades all the specified messages (files) of a selected group (e.g., all incoming mail), marking their original levels (and, optionally, other information such as their locations), and enters them into the secure subsystem, using a special discretionary marking. Typically, the markings designating the original level are entered into the file, and the entire file is put into a special format. The user then invokes whatever message system commands he wishes to process the messages. Each of these commands can access messages only through a secure subsystem routine which recognizes the special format and protects the security labels from alteration. When the user finishes processing the messages, he invokes a special downgrader that accesses the messages through another secure subsystem routine and automatically (i.e., without user intervention) reclassifies them, putting them back into their original format.

Security in this design is dependent upon a trusted set of verified subsystem routines that provide the sole access to the special format files (e.g., using the KSOS subtype manager). Applications software then can either move the file in toto or call the trusted routines to modify the file internally. For single-level messages, these routines may be fairly simple, consisting of a marker and upgrader, a reader, and a regrader. Employing these routines, message system application programs may contain commands to show and delete messages, scan headers, file and retrieve messages, etc. However, any operation that modifies messages or adds new information, such as composing a message or forwarding a message with annotations, requires either (a) that the user change levels and process the (new or changed) message at its appropriate level, or (b) that additional trusted subsystem routines be implemented to take a new piece of text, enter it into the secure subsystem at a level determined by the user, then append the new text onto a similarly marked message.

In this design, there is a question as to when to reclassify (downgrade) the marked messages. It would be inefficient to make the user reclassify all the messages in a folder each time he runs the message system. On the other hand, it may be costly to keep a substantial set of messages overclassified. One alternative is to have several upgrading strategies, applicable to various common situations and message sets (such as all incoming mail, or all folders for a particular project, etc).

The security policy heavily constrains the bookkeeping required for storing and retrieving messages in folders when upgrading and

reclassifying. The simplest method, albeit not the most efficient, is to create a separate mapping file with a map of message to folder. This map must be updated for most message system commands. It can be separate from the security level map and thus need not be trusted (although the data's integrity may be affected). Another alternative is to include this map in the secure subsystem, making it sufficiently trusted that message system commands at all levels could access and update it.

Because the marked messages are protected by the special secure subsystem, the user cannot access them with normal applications software. The user may of course read the messages, using the appropriate message system command, and create a file with the message that does not belong to the special subsystem (such a file would, of course, be protected by the host secure operating system). At that point, the message cannot be automatically reclassified and stored at the appropriate level with message system commands. The user will need to limit this type of access or risk creating an excess of overclassified files.

Functionality. Design 2.3 provides the same functionality as Design 2.2: The user can apply some processes (those that do not modify or create messages) to messages of several security levels at once, with the added benefit that the secure subsystem is responsible for reclassifying the messages. The user can manipulate multilevel folders and multilevel streams of incoming messages.

Ease of Use. As with the previous design, the user must tell the system when to move messages to a higher level for processing, and he

must be somewhat selective to avoid upgrading too many messages. In addition, the user may become confused as to whether a particular message (or file) is actually Top Secret, or is only temporarily stored as such to be later automatically downgraded by the secure subsystem.

Trust. This design violates the security model. It postulates an entire trusted security subsystem within the secure operating system, adding complexity to the system's security. The marking and reclassifying system has to be trusted, although the subsequent applications software does not, because it is not allowed to process the message directly.

Cost. Again, this design can store an excess of data at artificially high levels. Development costs include verifying the marking and regrading mechanisms.

Design 2.4: Protected context file.

When a user is processing a stream of messages at different security levels (e.g., his incoming mail), two key problems often recur:

1. In switching levels for a succeeding message (using the secure server), the user must deduce the appropriate next level, either noting that level from information supplied by the message system or searching for the level name in a message file.
2. The user must keep track of the context of the message or folder he was processing and reestablish that context in the new level.

This design attempts to deal with these problems using a context file that is accessible to processes at all security levels. It assumes that the user will handle each message by manually switching to a process of the appropriate level, but that the context of his actions will be available to the message system at the new level. Included in the context file are pointers to the current folder, the current message within that folder, and locations and levels of all folders. Thus, the user can read a message, switch to the level of the next message, and invoke a "next" command to read the next message in that stream (either from a folder or incoming messages). This design also assumes that processes can aid the user in switching levels by loading a buffer in the secure server with the right level name. Thus, the context file informs the user (and the secure server) which level to switch to and informs the message system processes at that level which message or folder to process.

Of course, a context file that all security levels can read and write violates the security model. Thus, this design depends upon a set of trusted routines to modify the context file, all of which are protected by a host system subtype manager. The modifications are further limited to certain types, e.g., choosing another already existing folder to become the current folder. The security level of the file itself is the lowest on the system, so that processes at any level can use it. Modifications to the file must be made through a protected routine, however. Because the file represents a confinement channel, it must be limited through the use of standard bandwidth-limiting techniques such as a long (e.g., 1-second) delay between invocations and a monitor to detect overuse by a single user.

The message system needs a mechanism to start up the right process after the user has switched security levels. Thus, when the user invokes a command, e.g., to read the next message in a message stream, and the security level of the next message is different, the message system can preload the secure server with the appropriate security level. In addition, the message system stores the "next" command in the context file. After the user changes levels, either he must start up the message system or it must automatically search for access to the terminal. In either case, the message system reads the command from the context file and invokes it.

There are a few special considerations with this design. Because the context file is unclassified, any logical names included in it, such as folder names, should be predefined (and unclassified), or some other means must be found to avoid passing classified information in the form of logical names to the context file. The context file itself is protected by the secure subsystem and therefore requires a special editor to modify it. If the applications software of the message system is not to be certified and trusted, the bandwidth of information to the context file must be limited and monitored; only certain changes will be legal, and only at a certain rate of change.

Functionality. The user can process a stream of messages as if they were of one level, except for having to press two or three keystrokes to switch levels with the secure server between some messages.

Ease of Use. Design 2.4 requires the user to change levels more often than do the previous designs. But changing levels involves a minimum of keystrokes (two or three), and the user is spared the cognitive trauma of constant, substantial interruptions. The security model is easy to understand, since the user's major operations (e.g., processing any one message) are within the operating system's security model.

Trust. The routines to modify the context file in this design violate the security model and must be verified. The amount of additional exposure is monitored and controlled by a confinement-channel monitor watching modifications to the context file. There is some question as to whether making level switching almost effortless jeopardizes the overall security of the system. The user may become cavalier about switching to levels with highly sensitive data, but this can be alleviated by making him aware (e.g., with colored lights or messages) of his current process level. Regardless of the user's awareness of the current security level, any information he creates will be protected at that level; the danger is that the user may forget that he has moved to a low classification level and then create messages (and other files) containing highly classified data.

Cost. Verifying the routines that modify the context file adds significantly to the development cost. Run time cost should be only slightly greater.

Design 2.5: Secure message system server.

This design postulates a message system server, similar to the KSOS secure server, that is able to start up processes at arbitrary levels and connect the user's terminal directly to these processes. This design is similar to Design 2.4 in that the user has a process at the same security level as the message he is handling, and he switches to a different-level process for messages of a different level. It differs in that the server itself automatically maintains the context file and interprets the message system commands to derive updates to that file. For example, if the user has just read a message at the Unclassified level and invokes a command to read the next message, which is at the Secret level, the server notes that the next message is Secret, starts (or restarts) a message-reading process at the Secret level, and points to the message of interest. It also updates the context file. Some sort of user confirmation may be advisable when the new level is different from the previous one.

Obviously, such a message system server must be verified. But this constitutes only a subset of the entire message system, the rest of which need not be verified. Only the processing necessary to maintain the context file and deduce the next target level are done in the server--the rest of the message processing is done in the individual (untrusted) processes invoked. Thus, the security of the system is greater than that of Design 2.6 below (all trusted code), and the development cost is substantially lower.

Functionality. The user can process a stream of messages, with the secure message system server automatically switching security levels for him.

Ease of Use. Extensive switching from level to level may distract the user with level-notification messages. The server controls the level changes for him, assuming his direct responsibility for level control. This may tend to reduce the user's awareness of the actual level, however, and may induce violations through carelessness.

Trust. This design violates the security model and violates the KSOS design of having only one server.

Cost. The secure message system server must be developed and verified. Further, the consequences of a multiserver design for the host operating system (e.g., KSOS) must be evaluated.

Design 2.6: Trusted code.

In a sense, this design is at the opposite extreme from Design 1.1: In that design, no (applications) code was trusted, whereas here all code must be trusted. The applications software is privileged to violate the underlying security model in any way. The user logs in at the highest level of data that he wishes to access and then starts the (trusted) message system. The trusted code is responsible for keeping the user informed and aware of the security level and the effects of his operations (e.g., downgrading). The trusted code can be accessed only through a secure server, to preclude the introduction of subversive code that mimics the trusted code. The host operating system's security features are still necessary, since messages will be stored in the file system and must still be protected from other users at different security levels. The cost of the high functionality of this class of designs is that all applications software must be trusted; it either

must be formally verified, or some other compelling evidence must be found for concluding that it will not compromise classified information.

Functionality. The functionality of this system is unlimited. The applications system can be programmed to give the user every convenience he could obtain from a message system in an unclassified environment.

Ease of Use. The ease of using this system depends upon the specific software implementation; it is potentially very high because of the lack of security constraints on software features.

Trust. The system is privileged to violate the security model completely. Each bit of code must be trusted and have its own evidence for support. All threats--e.g., Trojan horses, accidental disclosure, intentional penetration--are viable and cannot be prevented by the host operating system. In a real sense, the operating system security has been breached with a system that has all the faults that the operating system was designed to guard against.

Cost. Verifying or certifying large-scale, high-capability applications software as trustworthy by today's formal methods can range from prohibitively expensive to technically infeasible. Furthermore, all life-cycle modifications will require recertification.

3. Separation of Messages into Headers and Bodies at Different Security Levels

The third category of multilevel secure message processing systems incorporates all of the capabilities of the second category (i.e., multilevel folders and multilevel message streams) and extends it to accommodate messages in which the header and body may be at different security levels.

As with the previous category, it is implicitly assumed that the security features of the host operating system will be used to protect each message header or message body at its particular level and that the granule of protection at the system level is the file (e.g., Secret messages headers will be stored in files whose security level is Secret, etc.). A multilevel message must therefore have pointers to the different files the parts are stored in. In KSOS, a message could be implemented as a subdirectory containing a header file and a body file. Special trusted software must be implemented to move messages because files of different levels must be read and rewritten.

Design 3.2: Manual upgrade and downgrade.

In this design, both the header and body are upgraded to the highest level of messages being processed. The upgrading software has an additional bookkeeping task, that of marking the header and body with their original levels so the user can remark them correctly. The amount of effort required by the user to downgrade both header and body to their original levels might well make this approach impractical.

Design 3.3: Upgrade with protected markings, automatic downgrade.

Again, much as in Design 2.3, both the header and body are upgraded to the highest level of messages being processed; extra bookkeeping is required to separately track the message headers and bodies. In this design, one file could contain all of the fields for a particular message, marked by separate security labels. Then an additional set of routines protected by the secure subsystem is required

to extract and replace individual fields. All other message system software must use these trusted routines to access the message.

An additional cost and degradation of trust would occur due to the need to develop trusted code to handle the additional parsing and bookkeeping of level markings for internal parts of messages.

Design 3.4: Protected context file.

In Design 2.4, the user switched processes (and security levels) whenever he wished to manipulate a message of a different security level. A context file was used to hold information necessary to move from one message, and one level, to another. The context file informed the user (and the secure server) of the target level and told the message system at that level which message or folder to process.

For Design 3.4, we extend the context file by adding pointers for the header and body files of the message. The user handles each message by switching to a process at the level of the message part. Thus, he may have to switch levels between the header and body of the message. As in the previous category, the secure server could be preloaded with the correct level to minimize the number of keystrokes required to move between header and body.

Operations on the entire message, such as moving or deleting it, are difficult. The user must delete whatever part he can access (which will be noted in the context file) and then switch levels to delete the remainder. Similarly, to move a message into a folder, the user must move each part separately. Additional bookkeeping aids in the context file could assist these multistep operations, e.g., as preloaded

commands in the context file that the message system would automatically invoke when started at the succeeding security level. But with the additional level switching between parts of a message, the user may become confused about the security levels of the message parts.

Design 3.5: Secure message system server.

This design differs little from Design 2.5. The only difference is that the context file that the message system server must maintain is more complex and adds to the complexity of the server, which must be verified; the added complexity could add greatly to the cost of verification. For example, when the user reads a message, the server must track whether the user is reading the header or body and switch processes to the correct level to read the remainder of the message. As with Design 3.4, operations on entire messages, like moving and deleting messages, either require special trusted software or require switching processes to move or delete each message part.

Design 3.6: Trusted code.

This design is substantially the same as Design 2.6. The message system is larger and more complex, however, so the problems of software verification may be more severe.

4. Messages with Fields and Paragraphs at Different Security Levels

The fourth category of multilevel secure message processing designs is quite similar to the third, the difference being one of finer granularity and therefore greater software complexity. These designs

incorporate all of the capabilities of the third category (i.e., multilevel folders and multilevel message streams) and extend them to handle multilevel messages in which individual fields within the message and individual paragraphs within the message body may be at different security levels. In addition, one design in this category provides granularity at the level of individual words.

As with the previous categories, our initial assumption is that the operating system security system will be used to protect each message field or message body paragraph at its particular level. A multilevel message must contain these message fields and paragraphs or have pointers to the different files they are stored in. In KSOS, a message could be implemented as a subdirectory containing files of fields and paragraphs, since the granularity of the operating system is at the level of the file.

However, as the granularity of the objects protected by the security system becomes finer, a question arises as to whether the operating system security granule (individual files in KSOS) should be used to protect the security of each user-level data item that has a different security level. There are three alternatives:

1. The unit of protection in the host operating system security system maybe used to protect individual data items. In KSOS this would imply a separate file for each word, paragraph, or set of these. This same consideration applies to future operating systems which may more efficiently protect fine-grained user-level units of information (i.e., words or paragraphs). For our purposes,

the corresponding system-level units are equivalent in status to the file in KSOS; hence, we refer to KSOS files in our designs, though these often will not function as efficiently as may be required. Within this alternative, there are two possible implementations:

a. Each data item is stored in a separate file (or system-level security granule).

b. All data items from the same message and having the same security level are stored in a single file and are marked as to their relative location in the original message. For example, a message body with paragraphs 1, 2, and 5 classified Secret and paragraphs 3 and 4 Unclassified would be stored in two files--a Secret file containing paragraphs 1, 2, and 5, and an Unclassified file containing the remaining paragraphs. This kind of implementation requires software to interleave and separate out the individual items when translating into a representation more suitable for human processing (e.g., editing). In this implementation, the integrity of the entire document is in danger because the markings might be changed, i.e., the ordering markings on the paragraphs could be altered to shuffle the locations of the paragraphs in the overall document.

2. The data items may be protected by a secure subsystem, perhaps based on protected markings within a file (or other granule). In this case, the data items would have to be translated in and

out of the protected format before and after processing by most applications software.

3. Protection of the individual data items within a message may be the user's responsibility. This may be a viable alternative because the boundary between messages is a weaker bond than the boundary between parts of a message or a report. The user may seldom wish to separate parts of a message and will often want to manipulate the entire message. In this case, system-level granules (KSOS files) are classified at the level of the most highly classified pieces contained in them.

Design 4.2: Manual upgrade and downgrade.

In a design of this type, all of the message fields are upgraded to the highest level of messages being processed. The upgrading software has the additional bookkeeping task of marking each field with its original levels so the user can reclassify correctly. The message fields are stored originally in separate files and are stored as highest-level files during message processing.

It appears that downgrading many pieces of a message or document would be so time-consuming as to make this design infeasible. Instead of reliably downgrading the overclassified information, the user would probably tend to leave many messages at artificially high levels of classification.

Design 4.3: Upgrade with protected markings, automatic downgrade.

Again, all the message fields are upgraded to the highest level of messages being processed, and extra bookkeeping is required for each.

In this case, a single file can contain all of the fields for a particular message, marked internally with security labels. One additional set of trusted routines protected by the secure subsystem is required to extract and replace individual fields. All other message system software must use these routines to access the message.

The additional maintenance of security labels within files for internal parts of messages requires the development of verified code, which adds to the cost of this class of design.

Design 4.4: Protected context file.

In Design 3.4, the user switched processes (and security levels) whenever he wished to manipulate a message field of a different security level, and a context file passed all the necessary information to support the move from one message field, and one level, to another. The context file informed the user (and the secure server) which level to switch to and informed the message system at that level which message field, message, and folder to process.

Design 4.4 adds to the context file a pointer to any item of data in the message. The user handles each message by switching among processes at the levels of the various items. He may have to switch levels between fields, or even individual words, in the message. The secure server can be preloaded with the next correct level, requiring a minimum of keystrokes for each move. However, changing levels more than a few times in any particular message will certainly distract the user from his task to some extent.

Operations on the entire message, such as moving or deleting the message, are more difficult for this category, especially if the message is fragmented by levels. The user must manipulate whatever part he can access (which will be noted in the context file) and then switch levels to manipulate the remainder. Additional bookkeeping aids in the context file could assist in these multistep operations, but this design rapidly becomes unworkable for fragmented messages, due to added user effort and the potential for confusion.

One advantage to this design, however, is that each data item can be manipulated separately by untrusted applications software. For example, the context file might contain a pointer to a message to be edited. The user could edit a message part (all at the same level) by simply invoking an editor on that one part. A multilevel editor would read the context file for a pointer to the item to edit, call the untrusted single-level editor, and update the context file when the user was done editing that particular data item. To move to the next item, the user would have to switch levels, but the context file would already contain the editor's context at the new level. Obvious drawbacks to this design are the computational delays of process switching and editor startup, user effort to switch levels, user confusion, and delays incurred in limiting the context file confinement-channel bandwidth.

Design 4.5: Secure message system server.

This design differs little from Design 3.5. Here, the context file that the secure message system server must maintain is more

complex; this increases the complexity of the server, which must be verified. As with Design 4.4, operations on entire messages (e.g., moving or deleting messages) require either special trusted software or switching processes to move or delete each message part.

This design also permits the use of an editor that consists mainly of untrusted software, with an internal, protected context file that keeps track of which data are being edited. As with Design 4.4, the cost of process switching, in time and cycles, is large; but user effort to switch levels disappears, as does the confinement-channel limitation. This should greatly reduce the potential for user confusion, but it could lull the user into dimmed awareness of the current level. This scheme seems especially practical for editing documents with large blocks of text at the same security level.

Design 4.6: Trusted code.

On the surface, this design is substantially the same as Design 2.6. Of course, this fine-grained message system is larger and therefore even more difficult to verify.

However, some headway on workable designs for this design might be made by building a set of kernel primitives that each have restricted functions on the data but are verifiable. Thus, this design is not so much a system design as an applications software design strategy to limit the amount of code that needs to be verified.

In developing an editor, for example, the strategy leads to the following result: A trusted MAPPING-IN routine reads individual files of different security levels into core, constructing a separate bit map

defining the security level of each character of text. A set of trusted DELETE and MOVE functions are defined which can manipulate the character file but which also must manipulate the bit map. A trusted INSERT routine asks the user for the security level of new text about to be typed, then enters the new text into both the text-image and the bit map. Finally, a trusted MAPPING-OUT routine writes the text on to separate files by security level according to the bit map. Additional functions could be defined using the above as primitives; these additional functions need not be verified as to whether they preserve the mapping, although they do need to be trusted not to access the text or mapping except by means of the primitive functions.

Such a design, incorporating a kernel of primitives, may be simpler (and hence much less costly) to verify and trust but would probably require that the user review and verify the various security levels of the output text. The editor does have the substantial advantage of maintaining the security markings on the text instead of requiring the user to maintain them.

REFERENCES

Ames, S. R., and D. R. Oestreicher, "Design of a Message Processor System for a Multilevel Secure Environment," Proceedings of 1978 NCC, Vol. 47, AFIPS, 1978, pp. 765-771.

Bell, D. E., and L. J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, The MITRE Corporation, M74-244, Bedford, Massachusetts, October 1974.

Biba, K. J., Integrity Considerations for Secure Computer Systems, The Mitre Corporation, MTR-5153, Bedford, Massachusetts, June 1975.

Borden B., R. S. Gaines, and N. Z. Shapiro, The MH Message Handling System: User's Manual, The Rand Corporation, R-2367-AF, November 1979.

Defense Intelligence Agency, NMIC Support System User Manual, Planning Research Corporation, March 1979.

DeMillo, R. A., R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," CACM, Vol. 22, No. 5, May 1979, pp. 271-280.

Department of Defense, ADP Security Manual, Department of Defense Manual 5200.28-M, January 1973.

-----, Security Requirements for Automatic Data Processing (ADP) Systems, Department of Defense Manual 5200.28, December 1972.

Dion, L., "A Complete Protection Model," Proceedings, 1981 Symposium on Privacy and Security, April 1981.

Lampson, B. W., "A Note on the Confinement Problem," CACM, Vol. 16, No. 10, October 1973, pp. 613-615.

Landwehr, C., "Assertions for Verification of Multilevel Secure Military Message Systems," ACM Sigsoft Software Engineering Notes, July 1980.

Lipner, S. B., "Comment on the Confinement Problem," Operating Systems Review, Vol. 9, No. 5, May 1975, pp. 192-196.

McCauley, E. J., et al., Kernelized Secure Operating System--System Specification, TRW, Redondo Beach, California, April 1978.

-----, "KSOS: Design of a Secure Operating System," Proceedings of 1979 NCC, Vol. 48, AFIPS Press, New York, June 1979, pp. 345-354.

-----, Secure Minicomputer Operating System (KSOS) System Specification (Type A), 1978.

Miller, J. S., and R. G. Resnick, "Military Message Systems: Applying a Security Model," Proceedings 1981 Symposium on Security and Privacy, April 1981.

Myer, T. H., and C. D. Mooers, Hermes Message System User's Guide, Bolt, Beranek, and Newman, Inc., June 1976.

Secure Minicomputer Operating System (KSOS) System Specification (Type A), Ford Aerospace and Communications Corporation, WDL-TR7808, Rev. 1, Palo Alto, California, July 1978.

Walker, S. T., "The Advent of Trusted Operating Systems," Proceedings of 1980 NCC, Vol. 49, AFIPS Press, Arlington, Virginia, 1980, pp. 655-665.

Ware, W. H. (ed.), Security Controls for Computer Systems, The Rand Corporation, R-609-1, reissued October 1979.